

Hybridizing Non-dominated Sorting Algorithms: Divide-and-Conquer Meets Best Order Sort

Margarita Markina Maxim Buzdalov

April 14, 2017

Abstract

Many production-grade algorithms benefit from combining an asymptotically efficient algorithm for solving big problem instances, by splitting them into smaller ones, and an asymptotically inefficient algorithm with a very small implementation constant for solving small subproblems. A well-known example is stable sorting, where merge-sort is often combined with insertion sort to achieve a constant but noticeable speed-up.

We apply this idea to non-dominated sorting. Namely, we combine the divide-and-conquer algorithm, which has the currently best known asymptotic runtime of $O(N(\log N)^{M-1})$, with the Best Order Sort algorithm, which has the runtime of $O(N^2M)$ but demonstrates the best practical performance out of quadratic algorithms.

Empirical evaluation shows that the hybrid's running time is typically not worse than of both original algorithms, while for large numbers of points it outperforms them by at least 20%. For smaller numbers of objectives, the speedup can be as large as four times.

1 Introduction

Many real-world optimization problems are multiobjective, that is, they require maximizing or minimizing several objectives, which are often conflicting. These problems most often do not have a single solution, but instead feature many incomparable solutions, which trade one objective for another. It is often not known *a priori* which solution will be chosen, as decisions of this sort are often recommended to be made late, as the decision maker can

learn more about the problem [1]. This encourages finding a set of diverse incomparable solutions, which is a problem often approached by multiobjective evolutionary algorithms.

In the realm of scaling-independent preference-less, and thus general-purpose, evolutionary multiobjective algorithms, three paradigms currently seem to prevail [1]: Pareto-based, indicator-based, and decomposition-based approaches. Although there exist well-known decomposition-based [22] and indicator-based [25, 27, 28] algorithms, the majority of modern algorithms are Pareto-based [4–6, 26].

Most Pareto-based algorithms belong to one of big groups according to how solutions are selected or ranked: the algorithms which maintain non-dominated solutions [3, 4, 13], perform non-dominated sorting [5–7], use domination count [9], or domination strength [26]. In this research we concentrate on non-dominated sorting, as some popular algorithms make use of it [5, 6].

Non-dominated sorting assigns ranks to solutions in the following way: the non-dominated solutions get rank 0, and the solutions which are dominated only by solutions of rank at most i get rank $i + 1$. In the original work [20], this procedure was performed in $O(N^3M)$, where N is the population size and M is the number of objectives. This was later improved to be $O(N^2M)$ in [6].

As the quadratic complexity is still quite large, both from theoretical and practical points of view, many researchers concentrated on improving practical running times [8, 11, 16, 19, 21, 23, 24], however, without improving the worst-case $O(N^2M)$ complexity. Jensen was the first to adapt the earlier result of Kung et al. [14], who solved the problem of finding non-dominated solutions in $O(N(\log N)^{\max(1, M-2)})$, to non-dominated sorting. This algorithm has the worst-case complexity of $O(N(\log N)^{M-1})$. However, this algorithm could not handle coinciding objective values, which was later corrected in subsequent works [2, 10]. A more efficient algorithm for non-dominated sorting, or finding *layers of maxima*, exists for three dimensions [17], whose complexity is $O(N(\log \log N)^2)$ with the use of randomized data structures, or $O(N(\log \log N)^3)$ for deterministic ones. However, whether this algorithm is useful in practice is still an open question.

A large number of available algorithms for non-dominated sorting opens the question of algorithm selection [18]. What is more, a family of $O(N^2M)$ algorithms for non-dominated sorting resembles a family of quadratic algorithms for comparison based sorting, and the $O(N(\log N)^{M-1})$ non-dominated sorting algorithms seem to take up the niche of $O(N \log N)$ sorting algorithms (such as mergesort, heapsort, and randomized versions of quicksort).

For comparison-based sorting, the quadratic algorithms are often much simpler and demonstrate better performance on small data, while asymptot-

ically better algorithms take over starting from certain problem sizes. If the latter algorithm is built using a divide-and-conquer scheme, it becomes possible to choose better algorithms for subproblems: if a subproblem, due to its size, can be solved faster using a quadratic algorithm, then it should be done, otherwise let the divide-and-conquer algorithm decompose the problem further. For example, most stable sorting algorithms from standard libraries are currently implemented using mergesort or TimSort, while for data fragments smaller than, for example, 32 in the current implementation of sorting in Java¹, the quadratic insertion sort algorithm, with the binary search lookup, is used.

This inspired us to apply the similar idea to non-dominated sorting. For the “outer” divide-and-conquer algorithm, we use the only available algorithm family of this sort [2, 10, 12]. For the quadratic algorithm to solve smaller subproblems, we adapt the Best Order Sort [19], as it was shown to typically outperform other quadratic algorithms. Our result is a hybrid algorithm which uses primarily the divide-and-conquer strategy and decides when to switch to Best Order Sort using a formula which depends on the number of points in the subproblem and the number of remaining objectives to consider.

This is a full version of the paper with the same name which was accepted as a poster to the GECCO conference in 2017.

The rest of the paper is structured as follows. In Section 2, we give the necessary definitions and describe the algorithms we put together: the divide-and-conquer algorithm and Best Order Sort. Section 3 describes our hybridizing approach, which includes the changes necessary to introduce to Best Order Sort to serve as the subproblem solver, and the analysis of preliminary experiments which established the formula used to switch between the algorithms. Section 4 gives the main body of our experimental studies, including their analysis. Section 5 concludes.

2 Preliminaries

In the following, we assume that all points are different, which enables us to name any unordered collection of points a *set*. This is not true in general, however, all equal points will receive the same rank, so implementations are free, depending on their need, to either discard a point if there is an equal one, or to keep all equal points in a same entity and run algorithms on these entities instead, or to work directly with equal points with some

¹http://grepcode.com/file_/repository.grepcode.com/java/root/jdk/openjdk/8-b132/java/util/TimSort.java

additional algorithmic care. None of these precautions change the worst-case algorithmic complexity.

2.1 Definitions

We use capital Latin letters to denote sets of points, as well as the global constants N (the number of points) and M (the number of objectives), while small Latin letters are used for single points, standalone objectives and rank values, and small Greek letters are used for mappings. The value of the i -th objective of a point p is denoted as p_i .

In the rest of the paper we assume, without losing generality, that we solve a multiobjective minimization problem with the number of objectives equal to M . In this case, the *Pareto dominance relation* is determined on two points in the objective space as follows:

$$\begin{aligned} a \prec b &\leftrightarrow \forall i \in [1; M] \ a_i \leq b_i \text{ and } \exists i \in [1; M] \ a_i < b_i \\ a \preceq b &\leftrightarrow \forall i \in [1; M] \ a_i \leq b_i \end{aligned}$$

where $a \prec b$ is called the *strict dominance* and $a \preceq b$ is the *weak dominance*.

Non-dominated sorting is a procedure which, for a given set P of N points in the M -dimensional objective space, assigns each point $p \in P$ an integer *rank* $\tau(p)$, such that:

$$\tau(p) = \max\{0\} \cup \{1 + \tau(q) \mid q \in P, q \prec p\}.$$

In other words, a rank of a point which is not dominated by any other point is zero, and a rank of any other point is one plus the maximum rank among the points which dominate it.

Following the convention from [15], we call the set of all points with the given rank r a *non-domination level* L_r :

$$L_r = \{p \in P \mid \tau(p) = r\}.$$

2.2 The Divide-and-Conquer Approach

The divide-and-conquer approach dates back to 1975, when Kung et al. proposed a multidimensional divide-and-conquer algorithm for finding the maxima of a set of vectors [14], which, in the realm of evolutionary computation, corresponds to the set of non-dominated points, or to points with rank zero. The complexity of this algorithm is $O(\min(N^2, N(\log N)^{\max(1, M-2)}))$, which we shorten to $O(N(\log N)^{M-2})$ for clarity.

This algorithm can be used to implement non-dominated sorting in the following manner: first we determine the points with rank zero, then we remove these points and run the algorithm again on the remaining points (which yields points with rank one), then we repeat it until no points left. However, the worst-case complexity of this approach is $O(N^2(\log N)^{M-2})$. In contrast, fast non-dominated sorting, shipped with the original NSGA-II of Deb et al. [6], has a better $O(N^2M)$ complexity.

The divide-and-conquer approach has been generalized to perform non-dominated sorting by Jensen [12], shortly afterwards the NSGA-II arrived. The algorithm from [12] solves the problem in $O(N(\log N)^{M-1})$, which is much faster for small values of M , as well as for large values of N , than fast non-dominated sorting. However, this algorithm was designed with an assumption that no two points have equal objectives, which is often not the case, especially in discrete optimization, and is known to produce wrong results when this assumption is violated. This problem was overcome by Fortin et al. [10], who proposed modifications of this algorithm to always produce correct results. The average complexity was proven to be the same, but the worst-case complexity was left at $O(N^2M)$. Finally, Buzdalov et al. [2] introduced further modifications to achieve the worst-case time complexity of $O(N(\log N)^{M-1})$.

We shall now briefly illustrate the working principles of this approach. At any moment of time, the algorithm maintains, for every point p , a lower bound on its rank $r'(p)$, which are initially set to zero. The reason for this lower bound can be explained as follows: at any moment of time, we have performed a subset of necessary objective comparisons, which impose approximations of ranks of the affected points. These approximations are of course lower bounds of the real ranks.

To ease the notation, in the following we do not use the term “lower bound of the rank”, as well as the r' symbol. Instead, we will say “current rank” for the current state of the lower bound of a certain point, which possibly coincides with the real rank, and “final rank” when we know that the lower bound coincides with the real rank.

One of the main properties of the algorithm is that whenever a comparison of p_m and q_m is performed for the first time, where p and q are points and $1 \leq m \leq M$ is the objective, then the following holds:

- for all objectives m' such that $m < m' \leq M$, it holds that $p_{m'} \leq q_{m'}$, that is, p weakly dominates q in objectives $[m'; M]$;
- the rank of p is known and final, that is, all comparisons necessary to determine the rank of p have already been done.

The top-level concept is the procedure $\text{HELPERA}(S, m)$, which takes a set of points S sorted lexicographically (where non-zero lower bounds are possibly known for some of the points from S) and makes sure all necessary comparisons between the objectives $[1; m]$ of these points are performed. This procedure is called only when all necessary comparisons of points p and q , such that $q \in S$ and $p \notin S$, have already been performed. To perform non-dominated sorting of a set P with M objectives, one should run $\text{HELPERA}(P, M)$.

For $m = 2$, it calls a sweep line based algorithm $\text{SWEEPA}(S)$, which runs in $O(|S| \log |S|)$, which we will cover later. If there are at most two points in S , it performs their direct comparisons and updates the rank of the second point if necessary. If all values of the objective m are the same in the entire S , it directly calls $\text{HELPERA}(S, m - 1)$. Otherwise, it divides S into three parts using the objective m : the S_L part with lower values, the S_M part with median values, and the S_H part with higher values.

It is clear that ranks of points in S_L do not depend on ranks of points in neither S_M nor S_H , and S_M also does not depend on S_H . The algorithm first calls $\text{HELPERA}(S_L, m)$, which results in finding the exact ranks in S_L , because all necessary comparisons with points from S_L on the right side and other points on the left side have been performed before this call.

Next comes the set S_M , but the ranks of these points still need to be updated using the set S_L (and nothing more). To do this, the algorithm calls another procedure, $\text{HELPERB}(S_L, S_M, m - 1)$, whose meaning is to update the ranks of points from the second argument using the first argument and objectives in $[1; m - 1]$. Then it calls $\text{HELPERA}(S_M, m - 1)$, as all other necessary comparisons have been done, and all values for the objective m are equal in S_M . It then proceeds with $\text{HELPERB}(S_L \cup S_M, S_H, m - 1)$ and finishes with $\text{HELPERA}(S_H, m)$.

The $\text{HELPERB}(L, H, m)$ procedure, as follows from the short description above, shall perform all the necessary comparisons between points $p \in L$ on the left and $q \in H$ on the right, provided that in objectives $[m + 1; M]$ it holds that $p \prec q$, and all ranks in L are final. For $m = 2$, it, again, runs a sweep line procedure $\text{SWEEPB}(L, H)$. If $|L| = 1$ or $|H| = 1$, a straightforward pairwise comparison is performed. If the maximum value of the objective m in L does not exceed the minimum value in H , it calls $\text{HELPERB}(L, H, m - 1)$. Otherwise, it chooses a median of the objective m in $L \cup H$ and then, similarly to HELPERA , splits L into L_L , L_M and L_H , and also splits H into H_L , H_M and H_H . Following the same logic as in HELPERA , it performs the following recursive calls:

- $\text{HELPERB}(L_L, H_L, m)$;

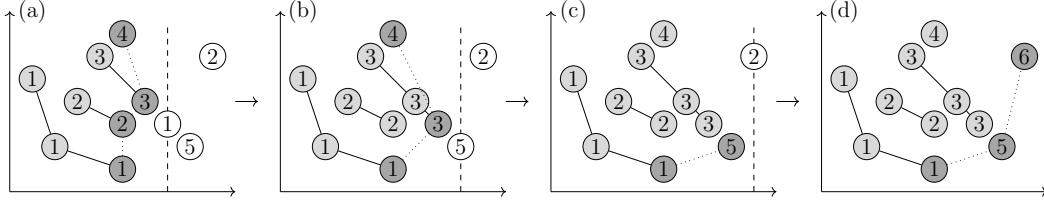


Figure 1: Example iterations of the SWEEPA procedure. Gray points are those whose rank is determined, the darker ones constitute a binary search tree and thus connected with dotted lines. The numbers in white points are the lower bounds on ranks. The vertical dashed line is the sweep line. In (b), the representer of level 2 is removed as all possible remaining points dominated by that point are also dominated by the representer of level 3; similar thing happens in (c).

- $\text{HELPERB}(L_L, H_M, m - 1);$
- $\text{HELPERB}(L_M, H_M, m - 1);$
- $\text{HELPERB}(L_L \cup L_M, H_H, m - 1);$
- $\text{HELPERB}(L_H, H_H, m).$

The remaining parts to explain are $\text{SWEEPA}(S)$ and $\text{SWEEP}(L, H)$. The SWEEPA procedure utilizes a sweep line approach. Points from the set S are processed in lexicographical order using first two objectives. In the same time, the procedure maintains a binary search tree which contains the last seen representative points for each non-domination level. When the next point is processed, this binary search tree is traversed to determine the biggest number of the level which still dominates the point in question, and then the rank of this point is updated correspondingly. After that, this point is inserted in the tree: it becomes the last representative of its non-domination level and possibly throws out some of the other representatives, which have no more chance to determine rank of any point on their own. An example is shown in Fig. 1.

The SWEEP procedure works in a similar way. The sweep line goes over the union of sets, $L \cup H$, however, the tree is built of the points from L only, and rank updates are performed with points from H only.

The running times of SWEEPA and SWEEP are $O(|S| \log |S|)$ and $O((|L| + |H|) \log |L|)$, respectively. From the well-known theory of solving recursive relations, and from the strategies of creating subproblems, it follows that the

running time of $\text{HELPERB}(L, H, m)$ is $O((|L| + |H|) \cdot (\log(|L| + |H|))^{m-1})$, and of $\text{HELPERA}(S, m)$ it is $O(|S| \cdot (\log |S|)^{m-1})$.

2.3 Best Order Sort

The Best Order Sort algorithm was proposed in [19]. It aims at removing as many comparisons to be performed as possible. To do this, it sorts all points by all objectives, thus constructing M sorted lists of points $L_1 \dots L_M$, and processes the points in the following order: first, all first points in the lists $(L_{1,1}, \dots, L_{M,1})$, then all second points $(L_{1,2}, \dots, L_{M,2})$, then all third points, et cetera, until every point is processed at least once.

When a point p is processed for the first time, assume it happens in the list of the m -th objective, its rank has to be determined. The key fact is that only the points which precede p in L_m can dominate p , because all other points have a greater value of the m -th objective. Thus, it makes sense to compare p with the points that precede it in L_m .

To further decrease the number of comparisons, it is worth noting that, when a certain point p is processed in objective m , all subsequent *new* points, that is the points which will be processed for the first time, will have a value of the m -th objective which is not smaller than the one of p . This means that the objective m can be safely removed from the list of objectives to test when some other point q is checked for being dominated by p .

The algorithm maintains a set of objectives to consider O_p for every point p . Initially, $O_p \leftarrow \{1, 2, \dots, M\}$. Whenever a point p is processed in the list of the m -th objective, it is removed from O_p . Whenever a point q is checked for being dominated by p , only the objectives from O_p need to be considered.

Finally, to determine the rank using fewer comparisons, the points, which have been already considered in each objective list and have been assigned ranks, are stored in separate lists, where each list corresponds to a rank. To determine the rank of the next point, one can perform either a linear scan (starting with rank zero and increasing ranks by one) or binary search for the rank. As the number of points in rank lists cannot be non-trivially bounded, both ways have the worst-case complexity of a single search of $O(LM)$, where L is the number of points in all lists.

Best Order Sort features two phases: the pre-sorting phase, which takes $O(NM \log N)$, and the domination scanning phase. The complexity of the latter, in the worst case, is $O(N^2M)$, but can be smaller under various conditions. For instance, when all points are non-dominating, the points have a chance to arrange such that the first N processed points are unique, which means that every such point is tested against $O(N/M)$ points in average, which results in $O(N^2)$ running time.

3 Hybridizing the Algorithms

Our hybridization scheme is similar to that of production-grade sorting algorithms tuned for performance. As the top-level algorithm, we use the divide-and-conquer algorithm. For each subproblem it decides, using certain heuristic, whether to continue using the divide-and-conquer strategy or to run Best Order Sort for this subproblem. In turn, Best Order Sort runs uninterrupted until it solves the assigned subproblem.

Two problems need to be solved for this scheme to work. First, the original Best Order Sort algorithm cannot be straightforwardly applied to solve subproblems, because subproblems may feature non-zero lower bounds for ranks of some points, which appear from comparisons of these points with other points, which are out of the scope of the current subproblem. It also does not support working with two point sets in order to serve as a back-end of HELPERB.

Second, the particular kind of heuristic to determine when to run Best Order Sort is unclear. The main problem with it is that it should have a low computation complexity: at most $O(N)$, because otherwise evaluation of this heuristic worsens the complexity of the divide-and-conquer algorithm. This means we cannot perform any complicated analysis, such as, for instance, principal component analysis, to predict which algorithm is best.

In this section we address these two problems, which determines the shape of our hybridization approach.

3.1 Adaptation of Best Order Sort

When working as a part of the divide-and-conquer algorithm, Best Order Sort can be called instead either HELPERA or HELPERB. In the first case, it needs to assign final ranks to a set of points S using first m objectives ($m > 2$, as SWEEPA, due to its simplicity, works faster than Best Order Sort under any conditions), provided that all other necessary comparisons have been already performed, and consequently every point p has a current rank $\tau(p)$, which is a lower bound of its real rank. The only difference to the original Best Order Sort is that some $\tau(p)$ can be non-zero. This is easily compensated by checking only rank lists with ranks greater than or equal to $\tau(p)$, and thus updating the rank only if the update is increasing.

The HELPERB case is slightly more involved. If Best Order Sort is called within HELPERB(L, H, m), then ranks of points from L are already known, and it is necessary to perform comparisons between points from L and points from H to update the current ranks $\tau(p)$ of points $p \in H$ using first m objectives. In this case, all points are merged and are processed altogether.

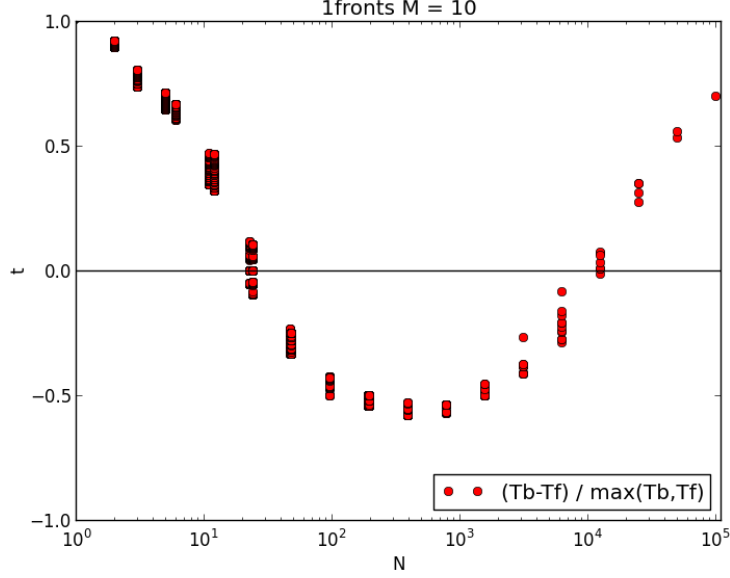


Figure 2: Example result of preliminary experiments on a dataset with 10 objectives and one non-domination level. The dataset has $N = 10^5$ points, all other points correspond to divide-and-conquer subproblems for this dataset. T_f is the running time of the divide-and-conquer algorithm, and T_b is the running time of Best Order Sort. The value of $(T_b - T_f) / \max(T_f, T_b)$ is plotted.

However, for points from L the rank is not updated (that is, the rank lists are never checked), instead they go directly to the corresponding rank lists. On the contrary, the rank update procedure is executed on points from H , but they are never added to rank lists.

These changes are quite small, so the correctness of Best Order Search in the changed conditions follows straightforwardly from the correctness of the original algorithm [19]. The worst-case complexity of the HELPERB case is $O(M \cdot |L \cup H| \cdot \log |L \cup H|, M \cdot |L| \cdot |H|)$.

3.2 Design of the Switch Heuristic

To understand the possible kind of the heuristic algorithm to use for deciding whether to use Best Order Sort for a certain subproblem, we conducted a series of preliminary experiments. In these experiments, we considered a series of datasets, where every dataset had $N = 10^5$ points with $M \in [3; 20]$ objectives and was generated either by uniformly random objective sampling (from the $[0; 1]^M$ hypercube) or by sampling from a hyperplane

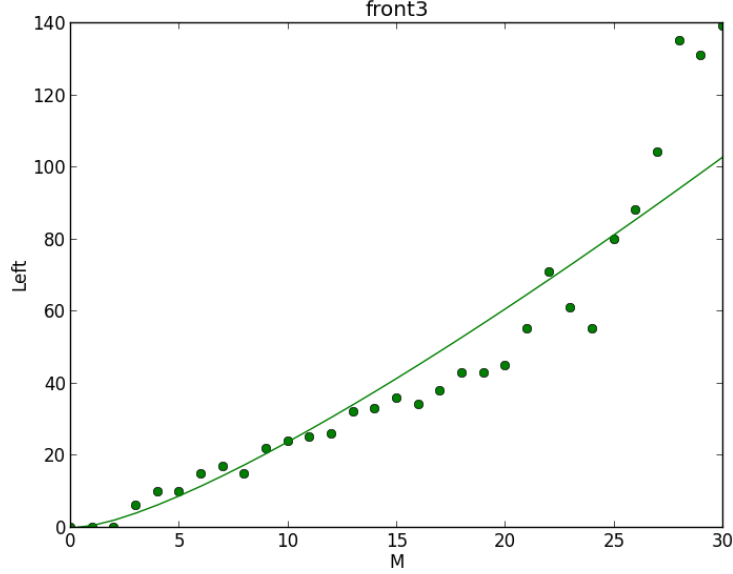


Figure 3: Left bounds of the BOS-efficient range: actual bounds from datasets with three levels and the fitted curve

(which yields a dataset with exactly one non-domination level). Then we ran the divide-and-conquer algorithm on each of these datasets and recorded all subproblems created during the run. After that, we measured the running times of both the divide-and-conquer algorithm and Best Order Sort on all these subproblems.

Fig. 2 shows an example of such experiment. In this figure, the point above the abscissa axis means that for the corresponding subproblem the divide-and-conquer algorithm took less time than Best Order Sort, while a point below zero means the opposite. One can clearly see in Fig. 2 that Best Order Sort behaves best, compared to the divide-and-conquer algorithm, for N which are not too small and not too large.

As the similar effect has been noticed for all other datasets as well, we attempted to deduce formulas for the left and right bounds of the higher efficiency range of Best Order Sort. The following empirically constructed formulas were found to fit our data rather well: $n_{\min} = m \ln(m + 1)$ and $n_{\max} = 150m((\ln(d + 1))^{0.9} - 1.5)$, where m is the current number of first objectives to consider, n_{\min} is the left bound of the range, and n_{\max} is the right bound. Fig. 3 shows the plot of the left bound formula and the actual left bounds in datasets with three non-domination levels, Fig. 4 does the same for the right bound formula and datasets with twenty levels. The fitting quality is the same for all other considered datasets.

As a result, the hybrid algorithm switches to Best Order Sort whenever

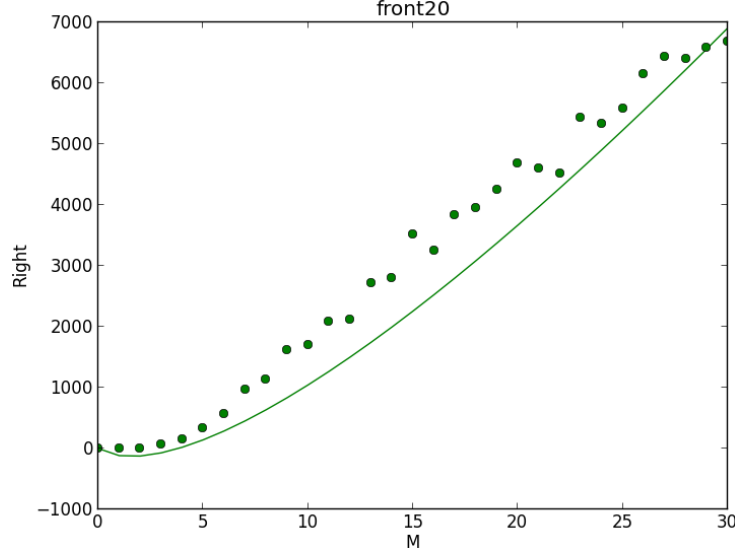


Figure 4: Right bounds of the BOS-efficient range: actual bounds from datasets with twenty levels and the fitted curve

the number of points n and the number of considered objectives m in the current subproblem satisfy:

$$m \ln(m + 1) \leq n \leq 150m \cdot ((\ln(d + 1))^{0.9} - 1.5).$$

4 Experiments

The main part of experiments was organized as follows. For every combination of:

- numbers of points $N = \lfloor 10^{n/4} \rfloor$ where $n \in [8; 20]$;
- numbers of objectives $M \in \{3, 5, 7, 10, 15, 20, 25, 30\}$;
- numbers of non-domination levels $L \in \{1, 2, 3, 5, 10, 20\}$;

ten datasets were created, and running times of all considered algorithms (the divide-and-conquer algorithm, Best Order Sort, and the hybrid algorithm) were measured.

The results are presented on pages 17–24 as bar plots for all M and L . Each bar plot features a section corresponding to the value of N , consisting of the following three bars: $T_{BOS}/\text{avg}(T_{DC})$, $T_{DC}/\text{avg}(T_{DC})$, $T_H/\text{avg}(T_{DC})$, where T_{BOS} is the running time of Best Order Sort, T_{DC} is the running

time of the divide-and-conquer algorithm, and T_H is the running time of the proposed hybrid algorithm. The bars for Best Order Sort are blue, and the bars for the hybrid algorithm are brown. Every bar has an average, minimum and maximum value (for the second bar plotting T_{DC} , the average is always one). Whenever a bar’s average is greater than one, that is, it points up, it means that the corresponding algorithm is slower than divide-and-conquer, and if it is faster, then the bar points down.

From plots on pages 17–24, one can immediately spot the characteristic behavior of Best Order Sort: it is typically better at smaller numbers of points, then it gradually becomes worse (for $M = 7$ and $M = 10$, this tendency is seen the best). For somewhat higher dimensions ($M = 20$ and $M = 25$), the lower bound of the Best Order Sort efficiency interval can be seen. For the highest considered dimension, $M = 30$, Best Order Sort demonstrates no significant improvement over the divide-and-conquer algorithm.

The hybrid algorithm tends to perform at least as good as the best of the two algorithms up to $M = 7$. Starting from $M = 10$, it features a somewhat suboptimal performance at the middle problem sizes ($N \in [10^3; 10^4]$) while still capturing the best behavior at small sizes and getting better than all other algorithms close to $N = 10^5$.

In fact, the hybrid is always better than its parts for big numbers of points. For $N = 10^5$, the average speedup compared to the best of the parts can be as large as 4.28 when $M = 3$, and never seen to get less than 1.198 in all other considered datasets.

5 Conclusion

We presented a hybrid algorithm for non-dominated sorting which initially runs a divide-and-conquer algorithm, however, when the size of a certain subproblem seems to be suitable, it solves this subproblem using another approach, Best Order Sort. For this to work, we slightly adapted Best Order Sort, so that it can perform non-dominated sorting in a more general setup, which needs to solve the divide-and-conquer subproblems. We also composed a heuristic rule for when to switch to Best Order Sort, which is based solely on the dimensions of a subproblem.

Our algorithm performs generally at least as well as its parts, except for certain ranges around the switchpoint between the algorithms at higher dimensions. This is an indicator that our heuristic on when to switch is not perfect yet and has a room for improvement. Nevertheless, for the wide range of testing data (3 to 30 objectives, 1 to 20 non-domination levels) our

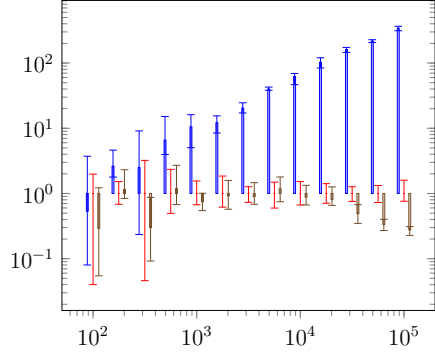
algorithm performs at least 20% better than the best of its parts for large numbers of points (such as $N = 10^5$), and the speedup can be up to 4x for smaller M . In a sense, this means that our hybridization scheme is rather robust.

References

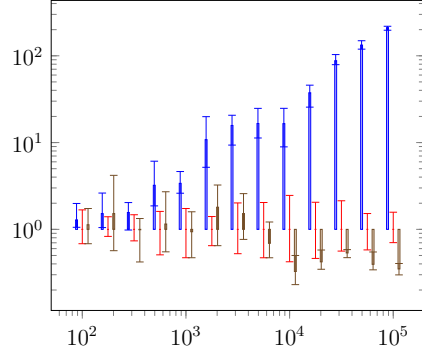
- [1] D. Brockhoff and T. Wagner. Gecco 2016 tutorial on evolutionary multiobjective optimization. In *Proceedings of Genetic and Evolutionary Computation Conference Companion*, pages 201–227, 2016.
- [2] M. Buzdalov and A. Shalyto. A provably asymptotically fast version of the generalized Jensen algorithm for non-dominated sorting. In *Parallel Problem Solving from Nature – PPSN XIII*, number 8672 in Lecture Notes in Computer Science, pages 528–537. Springer, 2014.
- [3] C. Coello Coello and G. Toscano Pulido. A micro-genetic algorithm for multiobjective optimization. In *Proceedings of International Conference on Evolutionary Multi-Criterion Optimization*, number 1993 in Lecture Notes in Computer Science, pages 126–140. 2001.
- [4] D. W. Corne, N. R. Jerram, J. D. Knowles, and M. J. Oates. PESA-II: Region-based selection in evolutionary multiobjective optimization. In *Proceedings of Genetic and Evolutionary Computation Conference*, pages 283–290. Morgan Kaufmann Publishers, 2001.
- [5] K. Deb and H. Jain. An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part I: Solving problems with box constraints. *IEEE Transactions on Evolutionary Computation*, 18(4):577–601, 2013.
- [6] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multi-objective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002.
- [7] M. Erickson, A. Mayer, and J. Horn. The niched Pareto genetic algorithm 2 applied to the design of groundwater remediation systems. In *Proceedings of International Conference on Evolutionary Multi-Criterion Optimization*, number 1993 in Lecture Notes in Computer Science, pages 681–695. 2001.

- [8] H. Fang, Q. Wang, Y.-C. Tu, and M. F. Horstemeyer. An efficient non-dominated sorting method for evolutionary algorithms. *Evolutionary Computation*, 16(3):355–384, 2008.
- [9] C. M. Fonseca and P. J. Fleming. Nonlinear system identification with multiobjective genetic algorithm. In *Proceedings of the World Congress of the International Federation of Automatic Control*, pages 187–192, 1996.
- [10] F.-A. Fortin, S. Grenier, and M. Parizeau. Generalizing the improved run-time complexity algorithm for non-dominated sorting. In *Proceedings of Genetic and Evolutionary Computation Conference*, pages 615–622. ACM, 2013.
- [11] P. Gustavsson and A. Syberfeldt. A new algorithm using the non-dominated tree to improve non-dominated sorting. *Evolutionary Computation*, Jan. 2017. Just Accepted publication.
- [12] M. T. Jensen. Reducing the run-time complexity of multiobjective EAs: The NSGA-II and other algorithms. *IEEE Transactions on Evolutionary Computation*, 7(5):503–515, 2003.
- [13] J. D. Knowles and D. W. Corne. Approximating the nondominated front using the Pareto archived evolution strategy. *Evolutionary Computation*, 8(2):149–172, 2000.
- [14] H. T. Kung, F. Luccio, and F. P. Preparata. On finding the maxima of a set of vectors. *Journal of ACM*, 22(4):469–476, 1975.
- [15] K. Li, K. Deb, Q. Zhang, and S. Kwong. Efficient non-domination level update approach for steady-state evolutionary multiobjective optimization. Technical Report COIN 2014014, Michigan State University, 2014.
- [16] K. McClymont and E. Keedwell. Deductive sort and climbing sort: New methods for non-dominated sorting. *Evolutionary computation*, 20(1):1–26, 2012.
- [17] Y. Nekrich. A fast algorithm for three-dimensional layers of maxima problem. In *Algorithms and Data Structures*, number 6844 in Lecture Notes in Computer Science, pages 607–618. 2011.
- [18] J. R. Rice. The algorithm selection problem. *Advances in Computers*, 15:65–118, 1976.

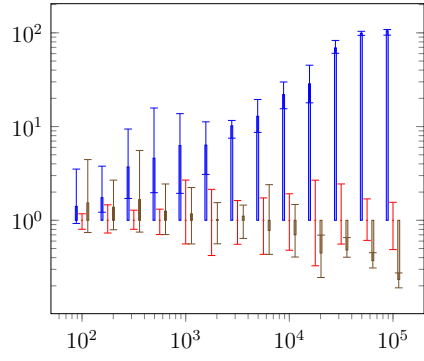
- [19] P. C. Roy, M. M. Islam, and K. Deb. Best Order Sort: A new algorithm to non-dominated sorting for evolutionary multi-objective optimization. In *Proceedings of Genetic and Evolutionary Computation Conference Companion*, pages 1113–1120, 2016.
- [20] N. Srinivas and K. Deb. Multiobjective optimization using non-dominated sorting in genetic algorithms. *Evolutionary Computation*, 2(3):221–248, 1994.
- [21] H. Wang and X. Yao. Corner sort for pareto-based many-objective optimization. *IEEE Transactions on Cybernetics*, 44(1):92–102, 2014.
- [22] Q. Zhang and H. Li. MOEA/D: A multiobjective evolutionary algorithm based on decomposition. *IEEE Transactions on Evolutionary Computation*, 11(6):712–731, 2007.
- [23] X. Zhang, Y. Tian, R. Cheng, and Y. Jin. An efficient approach to non-dominated sorting for evolutionary multiobjective optimization. *IEEE Transactions on Evolutionary Computation*, 19(2):201–213, 2015.
- [24] X. Zhang, Y. Tian, R. Cheng, and Y. Jin. A decision variable clustering-based evolutionary algorithm for large-scale many-objective optimization. *IEEE Transactions on Evolutionary Computation*, 2016.
- [25] E. Zitzler and S. Künzli. Indicator-based selection in multiobjective search. In *Parallel Problem Solving from Nature – PPSN VIII*, number 3242 in Lecture Notes in Computer Science, pages 832–842. 2004.
- [26] E. Zitzler, M. Laumanns, and L. Thiele. SPEA2: Improving the strength pareto evolutionary algorithm for multiobjective optimization. In *Proceedings of the EUROGEN’2001 Conference*, pages 95–100, 2001.
- [27] E. Zitzler and L. Thiele. Multiobjective evolutionary algorithms: A comparative case study and the Strength Pareto approach. *IEEE Transactions on Evolutionary Computation*, 3(4):257–271, 1999.
- [28] E. Zitzler, L. Thiele, M. Laumanns, C. M. Fonseca, and V. Grunert da Fonseca. Performance assessment of multiobjective optimizers: An analysis and review. *IEEE Transactions on Evolutionary Computation*, 7(2):117–132, 2003.



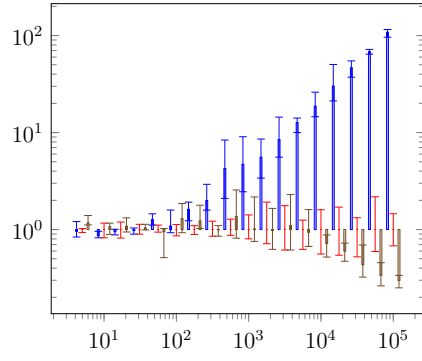
$M = 3$, one level



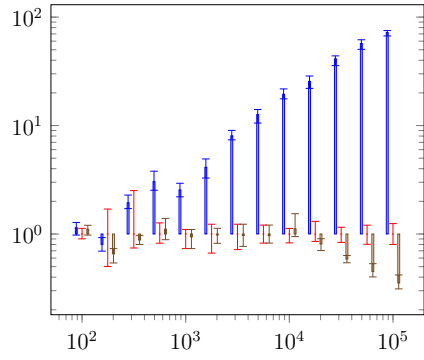
$M = 3$, two levels



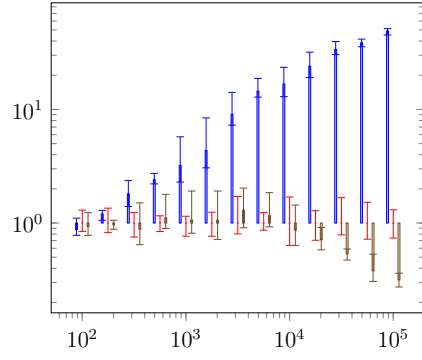
$M = 3$, three levels



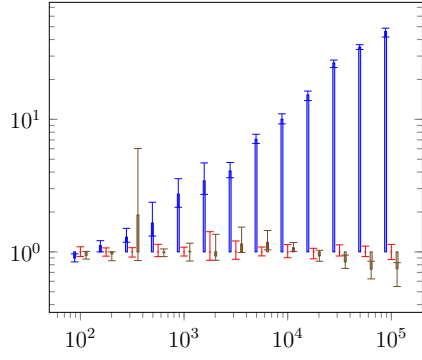
$M = 3$, five levels



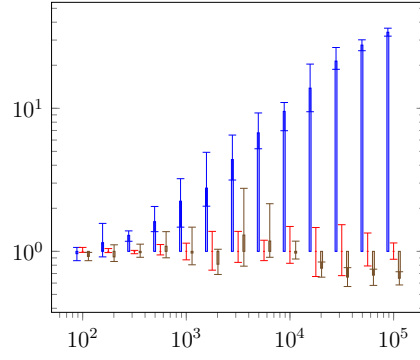
$M = 3$, 10 levels



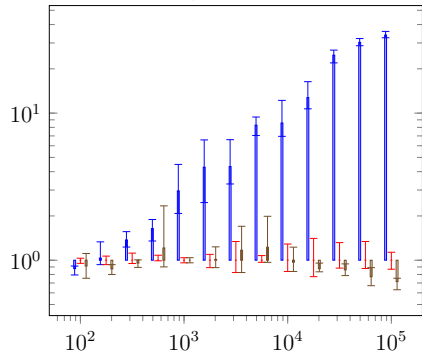
$M = 3$, 20 levels



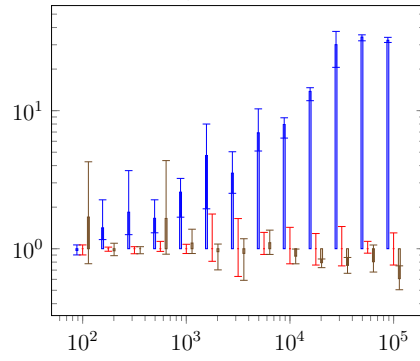
$M = 5$, one level



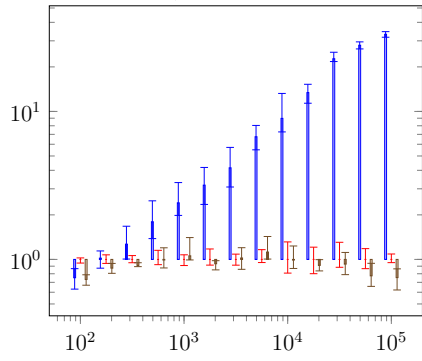
$M = 5$, two levels



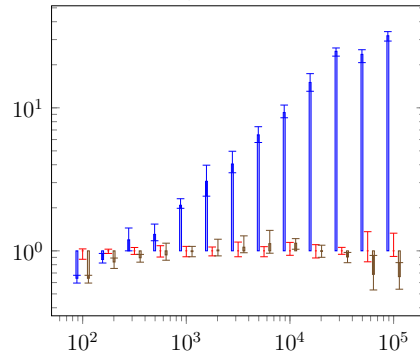
$M = 5$, three levels



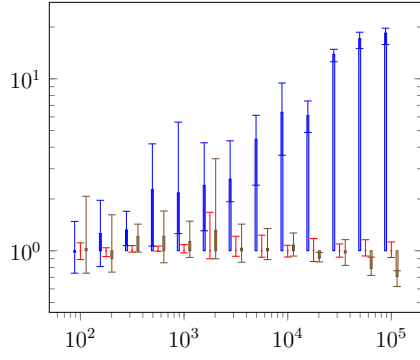
$M = 5$, five levels



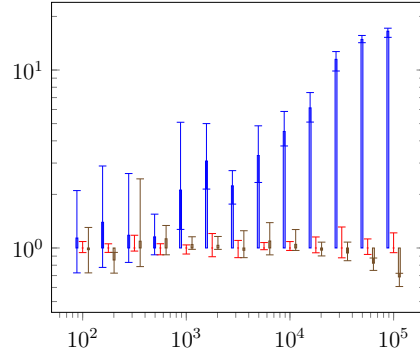
$M = 5$, 10 levels



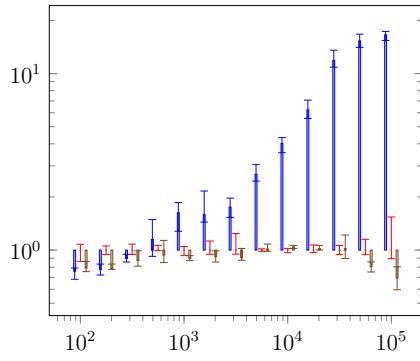
$M = 5$, 20 levels



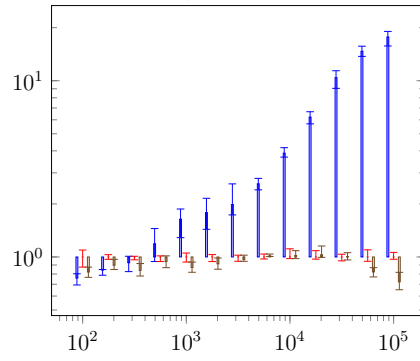
$M = 7$, one level



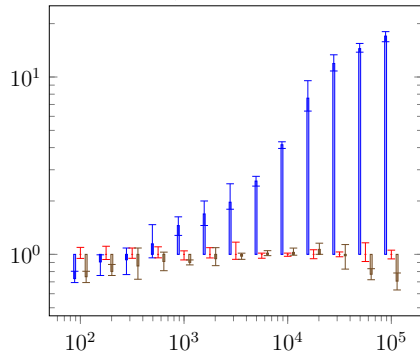
$M = 7$, two levels



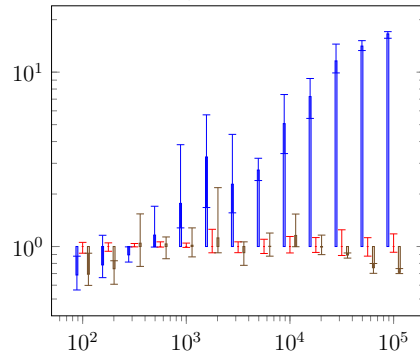
$M = 7$, three levels



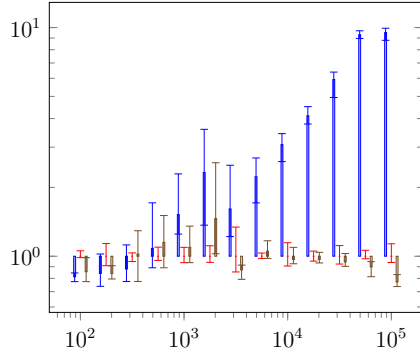
$M = 7$, five levels



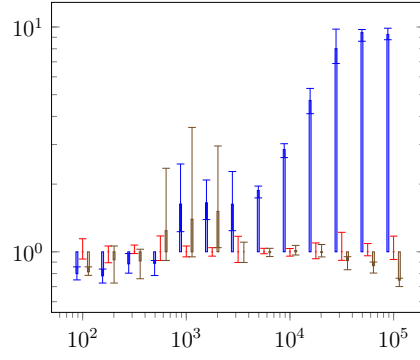
$M = 7$, 10 levels



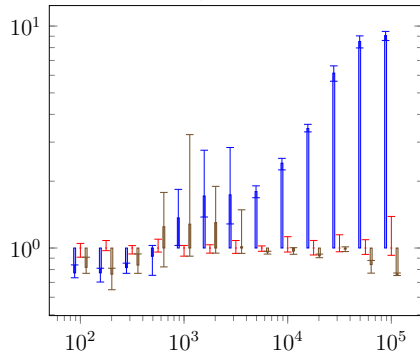
$M = 7$, 20 levels



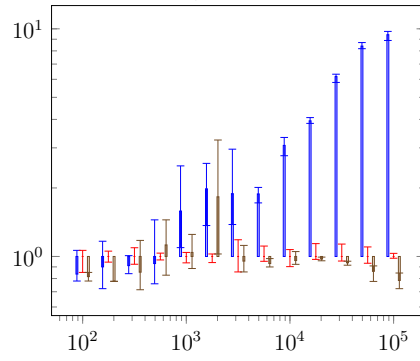
$M = 10$, one level



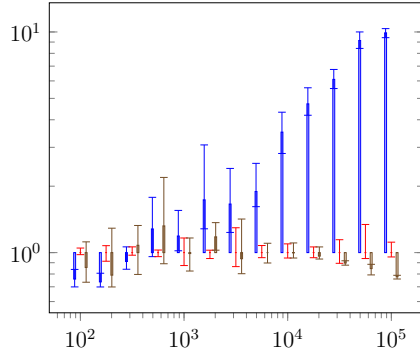
$M = 10$, two levels



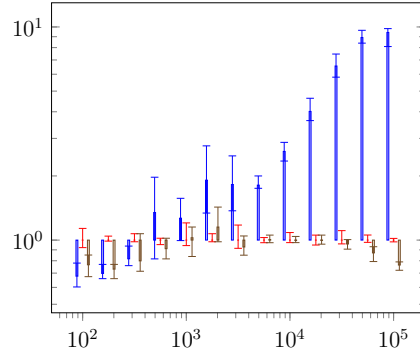
$M = 10$, three levels



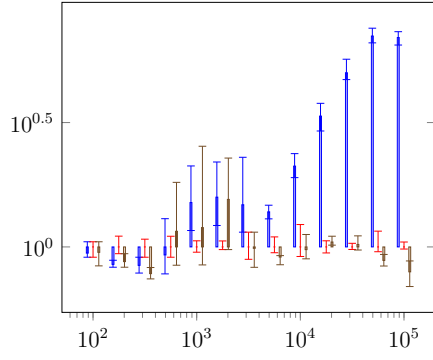
$M = 10$, five levels



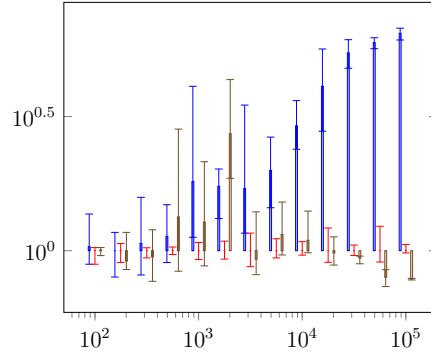
$M = 10$, 10 levels



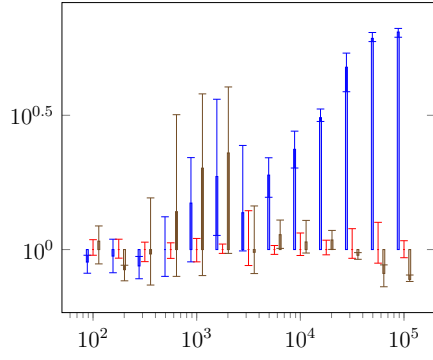
$M = 10$, 20 levels



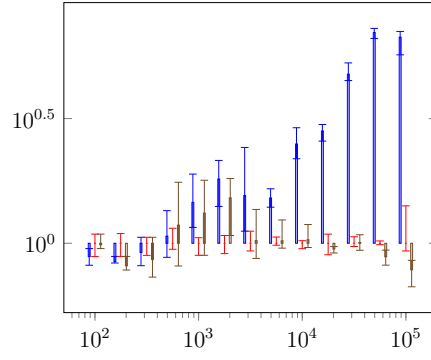
$M = 15$, one level



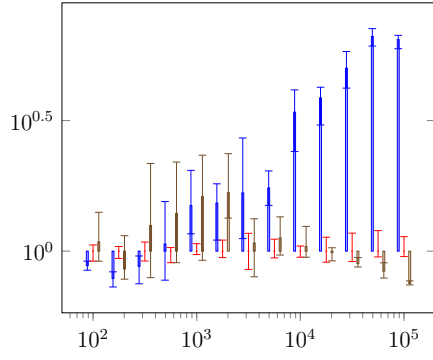
$M = 15$, two levels



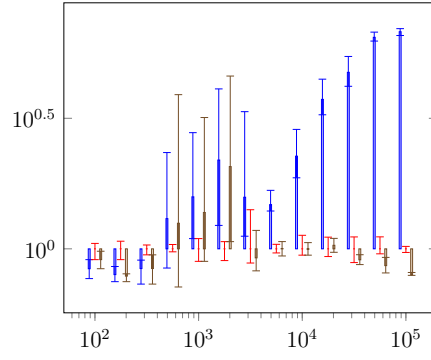
$M = 15$, three levels



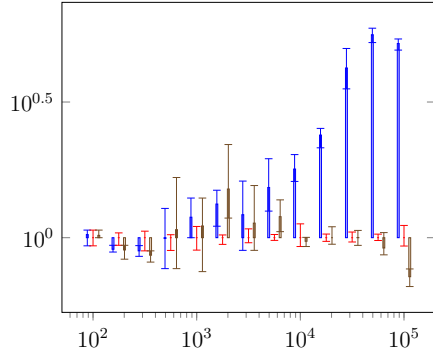
$M = 15$, five levels



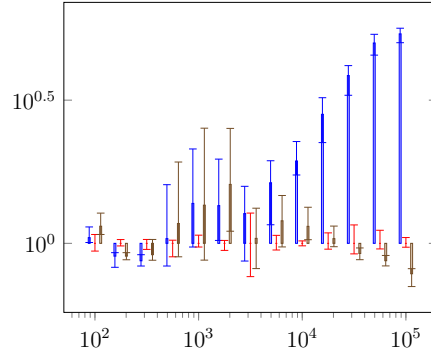
$M = 15$, 10 levels



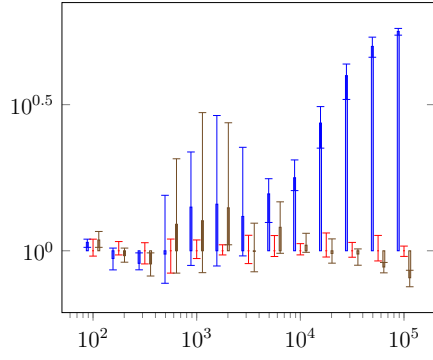
$M = 15$, 20 levels



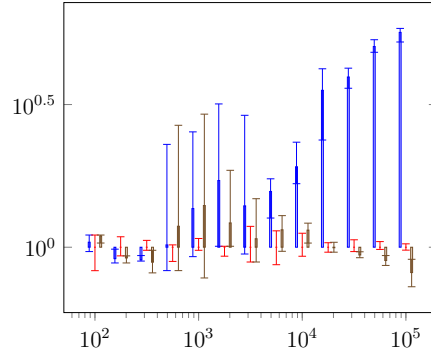
$M = 20$, one level



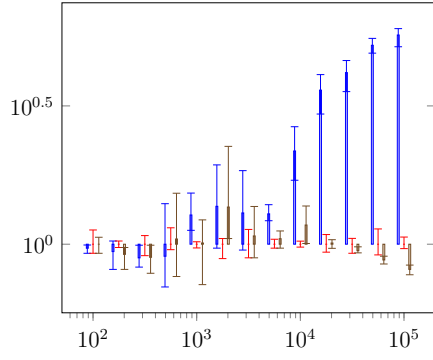
$M = 20$, two levels



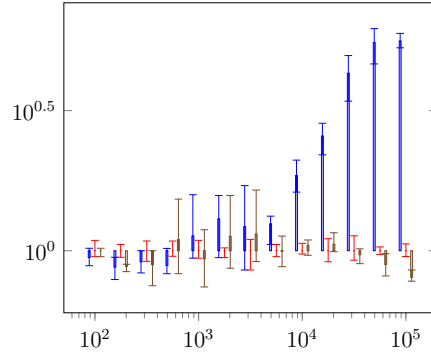
$M = 20$, three levels



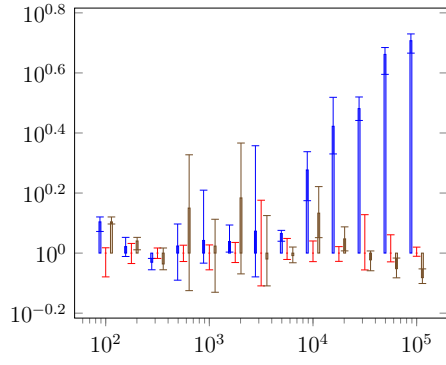
$M = 20$, five levels



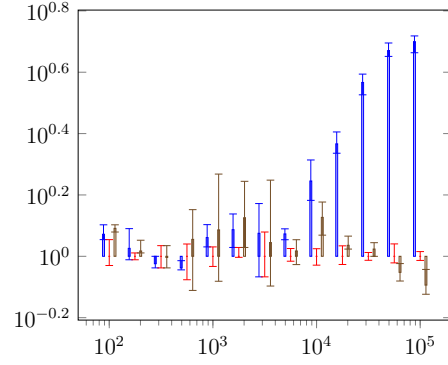
$M = 20$, 10 levels



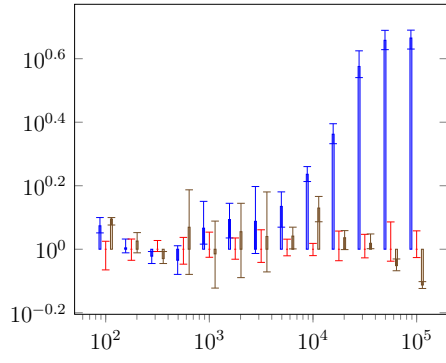
$M = 20$, 20 levels



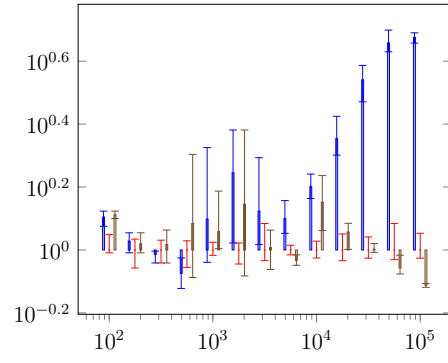
$M = 25$, one level



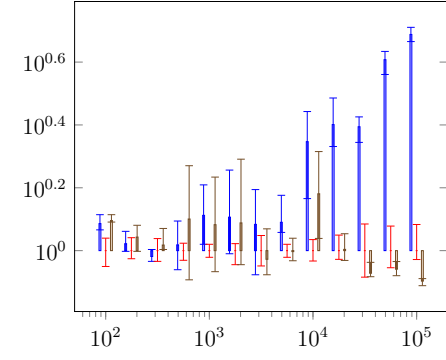
$M = 25$, two levels



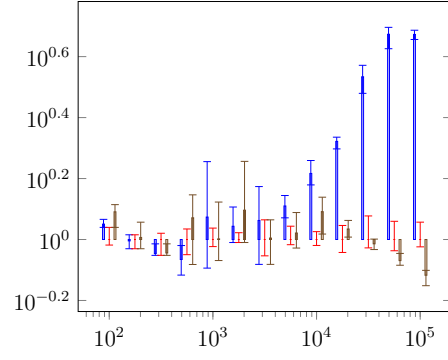
$M = 25$, three levels



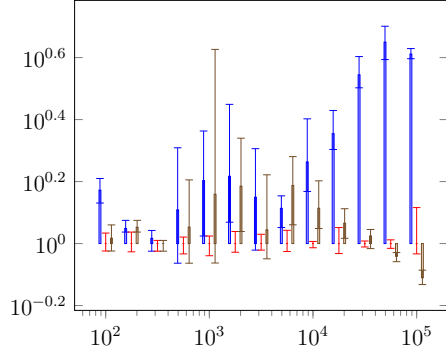
$M = 25$, five levels



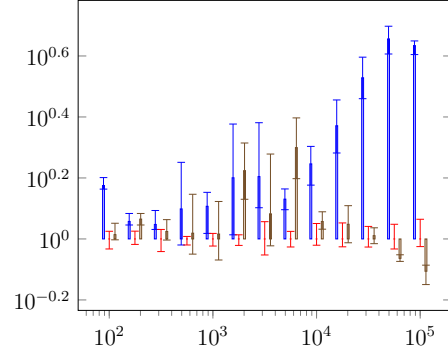
$M = 25$, 10 levels



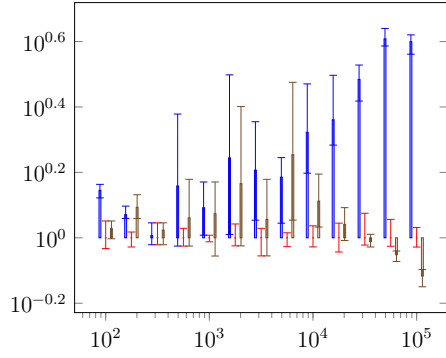
$M = 25$, 20 levels



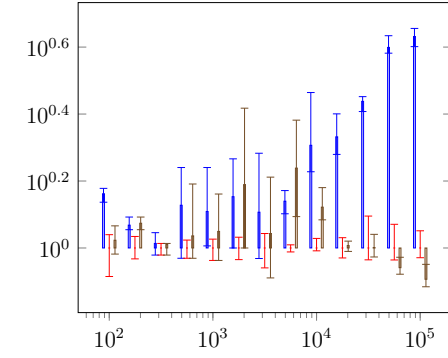
$M = 30$, one level



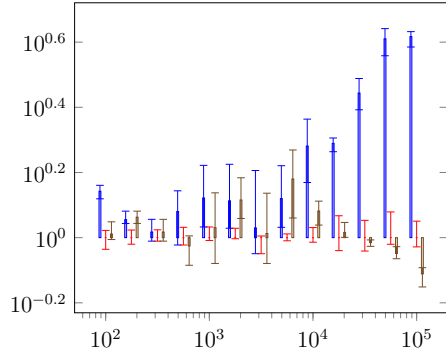
$M = 30$, two levels



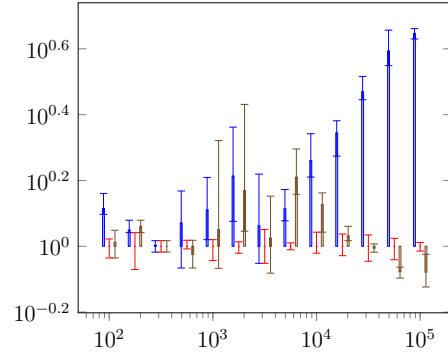
$M = 30$, three levels



$M = 30$, five levels



$M = 30$, 10 levels



$M = 30$, 20 levels